GETTING MORE OUT OF YOUR ATARI

by

Chris Crawford and Lane Winner

ATARI

1272 Borregas Ave

Sunnyvale, CA 94086

GETTING MORE OUT OF YOUR ATARI (R)

The ATARI 400/800™ is a second generation personal computer. In addition to the normal collection of RAM, ROM, and CPU, it contains three special purpose LSI chips which make it capable of many feats of computing legerdemain. Most of this power, however, lies brooding beneath many layers of human engineering. The beginning programmer working in pure BASIC is paternalistically protected from the complexities and power of the beast within. The more experienced programmer seeking cybernetic high adventure must first defeat the friendliness engineered into the machine to get at the brute power throbbing within. Without help, the task can be most difficult. In this article, we will act as native guides for one region of this complex machine: the display list. We will show you how to generate flashy displays by creating your own display list and redefining the character set.

DISPLAY LIST FUNDAMENTALS

Most personal computers use a straightforward memory-mapped display in which the screen format is fixed and each screen pixel's contents are provided by a specific location in RAM. This is a simple scheme demanding little of either the programmer or the computer. The Atari 400/800 uses a

more complex scheme involving a display list and display data. A display list is a sequence of commands which define the vertical format of the video display. The display data is the information to be displayed.

The display list is actually a small program; it is processed by a special LSI chip called ANTIC. ANTIC is a dedicated microprocessor whose sole function is to control the video display. ANTIC uses a process called DMA (direct memory access) to gain access to the display list and display data stored in RAM. The display list and display data are placed into RAM by the high speed (1.8 MHz) 6502 microprocessor. When the BASIC programmer types "GRAPHICS N" the operating system writes a complete display list into RAM and clears the display data. The information flow for this process is diagrammed in Figure 5. Clearly the adventurous programmer who bypasses BASIC and writes his or her own display list will have more direct control over the screen.

Associated with the display list are the concepts of a graphics mode and a graphics mode line. The Atari 400/800 supports fourteen fundamental graphics modes, only nine of which are directly accessible from BASIC. The first six modes (three of which are accessible from BASIC) are character modes which display characters in different size/color combinations. The remaining eight modes are graphics modes which display squares of color in different resolution/color combinations. A graphics mode line is a group of horizontal scan lines which

are treated as a unit for display purposes. (A horizontal scan line is a single sweep of the electron beam across the television screen. There are 192 horizontal scan lines in a complete screen.) A graphics mode line will contain between one and sixteen horizontal scan lines, depending on the graphics mode involved. A graphics mode line stretches horizontally all the way across the screen; one cannot change graphics modes halfway across the screen. The video display is thus organized as a vertical sequence of mode lines of varying height and contents. There are many thousands of possible sequences of mode lines on the screen; BASIC restricts the programmer to seventeen such sequences. Each such sequence is referred to in the BASIC manual as a graphics mode. The first part of this article will show how to create any sequence of mode lines down the screen.

DISPLAY LIST DETAILS

The display list and display data reside at the top of available RAM. Since their size can vary, the address of the beginning of the display list is stored in addresses 560 and 561 (decimal). Thus the start of the display list mode byte sequence, which we will assign to a BASIC variable called START, can be calculated by:

```
START=PEEK(560)+256*PEEK(561)+4
```

The bytes at this location and the succeeding location give the start address of the display data. Beginning at location START+2 is a sequence of mode bytes which specify the mode lines for the display. The codes for these mode bytes can be found in Table 1. The programmer has the freedom to create any sequence of mode bytes for the display list. The programmer also has the responsibility to insure that his or her chosen sequence includes exactly 192 horizontal scan lines. At the end of the mode byte sequence the programmer must place an ANTIC JUMP byte (decimal 65) followed by the low and high order address bytes of the beginning of the display list---four bytes lower in memory than the location we refer to as START.

The starting address of the display data, which we will assign to a BASIC variable called MEMST, can be calculated from:

```
MEMST=PEEK(START)+256*PEEK(START+1)
```

The display data is simply strung together in sequence; this can cause a headache when mixing modes. Since different mode lines require different numbers of display data bytes, the programmer wishing to change a display data byte must calculate its position in display data memory by adding up the RAM requirements of each previous mode line. The BASIC POSITION and PLOT commands only work reliably with the homogeneous display

lists used by BASIC, so the programmer who mixes modes must expend greater effort to use such a specialized display.

A REAL DISPLAY LIST

We shall now illustrate these principles with a sample program and its resultant display, display list, and display data. The program is a rather straightforward affair which plots the BYTE logo in graphics mode 7+16. It is presented in Listing 1, and the display it produces is shown in Figure 1. Figure 2 shows the display list for this display. Since this is a standard BASIC graphics mode display list, it is neat and tidy.

TAMPERING WITH THE DISPLAY LIST

We shall now tamper with this display list with the formal goal of improving the display and the heuristic goal of demonstrating display list manipulations from BASIC. The first step in this process is to prepare our proposed display list on paper.

We must consult Table 2 to determine which of the display modes will require the greatest amount of RAM. In our case, we are using modes 0, 1, 2, and 7; mode 7 is clearly the most RAM-intensive mode. We shall therefore start with mode 7 and modify the mode 7 display list. It is always easier to pare down an oversize display list than to build up an undersize one.

Next we must verify that our proposed display list does indeed produce 192 horizontal scan lines. We consult Table 1 to find the number of scan lines per mode line. Our calculation produces the following results:

MODE	NUMBER OF MODE LINES	SCAN LINES PER MODE LINE	TOTAL SCAN LINES	
0	1	8	8	
1	4	8	32	
2	4	16	64	
7	44	2	88	
			192	TOTAL

The next step in the procedure is unnecessary if the first mode line in the display list (the top line on the screen) is the same as the maximum RAM mode. If the first mode line is not the same as the maximum RAM mode, then we must set the upper nibble of the mode byte to 4 instead of 0. In our example, the first mode line is in graphics mode 7, so we leave the upper nibble at 0.

We now determine the mode bytes for each of the mode lines by simply looking them up in Table 1. We will find it handy to convert these to decimal for later use. Our results are:

MODE	HEX MODE BYTE	DECIMAL MODE BYTE
0	02	2
1	06	6
2	07	7
7	0d	13

The results of all this paperwork are presented on the right-hand side of Figure 2.

Now at last we are ready to write some code. Please refer to Listing 2 in conjunction with this narrative. We begin by checking if there is enough memory available to allow us to reposition the display list (line 0). If there isn't enough, the program aborts. We then move the top of available memory down 4K bytes and execute a GRAPHICS call (line 20) to write a new display list and display data in memory. This procedure reserves 4K of RAM for our own use later on. We then define our display strings (lines 30 and 40) and execute another GRAPHICS call to initialize our display list---which we shall subsequently modify. The series of POKEs in lines 50 and 55 define the colors we'll be using and turn off the character display while we redefine our characters. We then calculate the variable START in line 60. If the first mode line were not the same as the maximum RAM mode, we would then execute a POKE START-1,XX. XX is the decimal value of the mode byte with the upper nibble set to 4. In our case, however, the POKE is

unnecessary because the first mode line is the same as the maximum RAM mode. In lines 70-90 we POKE the new and different mode bytes into the display list to create our new display list. The offsets from START (the numbers added to START) are simply the mode line numbers for the new mode lines. Thus, the offset in line 70 is 10 because the mode byte we are POKEing is for the 10th mode line from the top of the screen. Remember, a mode line is not the same as a scan line. In line 95 we POKE the ANTIC JUMP byte and the jump address bytes at the end of our new display list. These bytes point to the beginning of the display list and can be found in locations 560 and 561.

We have just created a new display list on top of the original one. Now we must put a display onto the screen. This will be a tricky operation; as we mentioned earlier, the PLOT and POSITION commands will not quite work as we expect them to. Some extra finagling will be necessary to get a display up. Fortunately, our GRAPHICS 7 plotting of the Byte logo will still work the same way. Because we have inserted a mode zero line above it, the logo will be shifted down on the screen by six scan lines. This shift is so small that we can neglect it and plot the Byte logo with the same routine we used earlier. This is done in lines 110-120.

Now that we have plotted the logo we desire to print the characters. Two problems impede us: first, we must redefine the character set to mix upper- and lowercase characters, and second, we must calculate where these characters go.

The first problem arises from the natural limitations of an eight-bit processor. If four colors are supported (as in GRAPHICS 1 and 2) only 64 distinct characters can be displayed in each color. This is because two bits are required to specify the color, leaving only six bits to specify the character. This restricts our available set; the Atari character set in ROM supplies uppercase and punctuation or lowercase and graphics symbols but not uppercase and lowercase together---at least not in GRAPHICS 1 or 2. Since we want uppercase and lowercase together, we'll have to redefine our character set into RAM.

To do this we have to have some RAM reserved for our new character set. Line 20 did this by fooling the operating system into believing that the top of RAM (called RAMTOP) lies 16 pages lower than it actually does. This reserved 4K of RAM for our own use. The character set needs only 1K of RAM, but the display data cannot cross a 4K boundary, hence we must move the display list and display data down by an entire 4K. The address of the beginning of our new character set is calculated in line 200 and is called ADDR.

In line 210 we move the original character set in ROM starting at address 57344 into RAM. In line 220 we tell the operating system where the new character set is. In line 230 we move the uppercase characters into the positions previously occupied by punctuation. Our new 64 member character set has uppercase and lowercase, but very little punctuation. In line 240 we define a new space character, as the original space

character was part of the old punctuation group. We shall use the place previously occupied by the @ character for our space character.

We next take this technique of defining our own characters one step further. We had earlier decided to elevate the lowercase c in 'McGraw Hill'. To do this, we must redefine what a lowercase c looks like. This is done in line 250, with data coming from line 999. A graphical illustration of the logical process used to define the new character is provided in Figure 4. Obviously this procedure can be greatly extended. The diligent programmer can define any character set that can be expressed in an 8x8 grid and POKE it into RAM directly. Greek, Cyrillic, or special technical character sets can be created in this way.

We now have our display list and character set in order. We need only print our text. This is done starting at line 290. The first POKE suppresses the cursor for a neater display; the second POKE fools the operating system into believing that it is working in graphics mode 0. This prepares the way for a straightforward POSITION and PRINT of the first text line. The only trick is that the line is positioned vertically according to the number of mode lines from the top of the screen.

The next two text lines pose a particularly knotty problem. We desire to print GRAPHICS 1 and 2 characters on mode lines 46 through 52. Neither graphics mode allows so many lines; when we try to position the cursor onto line 46 the

computer will flip us a 'cursor out of range' error. Our only recourse is to POKE the character bytes directly into the display memory. We do this starting at line 310. First we calculate the starting address of the display memory (MEMST). Then we calculate the address where our characters are to be stored (CHRPOS). Our calculation relies on the fact that our characters are on the 46th line and all previous lines used 40 bytes each. In more complicated situations we would have to add up the byte requirements of all previous lines. This can get messy when a display mixes mode 1 and 2 lines at 20 bytes per line with other modes that use 40 bytes per line. Fortunately, our case is simple. Once CHRPOS has been calculated, we POKE the character values into the display data using a simple loop (line 320). Adding 60 to CHRPOS (line 330) skips 3 of our 20-byte mode 1 and 2 lines. We then POKE the character values for our third text line using the same technique that we used in line 320, except that a different character value offset (-64 instead of +128) gives us green characters instead of red ones. Line 350 turns the characters back on.

With this our program is complete; when it runs it produces the display shown in Figure 5.

CONCLUSION

The two major tricks we have demonstrated in this article (modifying the display list and redefining the character set) will greatly extend the graphics and display power of your

BASIC programs. The Atari 400/800 running in pure BASIC alone has stunning graphics capabilities. With these tricks, the machine brings previously unheard-of capabilities into the hands of the personal computer owner. Yet, we are still just trundling down the runway. There are even grander functions built into this machine---movable graphics objects for animation, vertical and horizontal fine scrolling, and display list interrupts, to name a few. With these tricks in hand, we can soar into the wild blue yonder of color display and animation.